Ultra-Fast Basic Geometrical Transformations on Linear Image Data Structure

Vladan Vučković[1], Boban Arizanović[2], Simon Le Blond[3]

[1]Faculty of Electronic Engineering, Computer Department, P.O. Box 73, 18000 Niš, Serbia, vladanvuckovic24@gmail.com

[2]Faculty of Electronic Engineering, Computer Department, P.O. Box 73, 18000 Niš, Serbia, bobanarizanovic@hotmail.com

[3]University of Bath, Department of Electronic & Electrical Engineering, United Kingdom, S.P.leBlond@bath.ac.uk

Abstract: This paper presents a general, ultra-fast approach for geometrical image transformations, based on the usage of linear lookup hash tables. The new method is developed to fix distortions on document images as part of a real-time optical character recognition (OCR) system. The approach is generalized and uses linear image representation combined with pre-computed lookup tables. To achieve maximal computational performance, pointer arithmetic and highly-optimized low-level machine code implementations are provided, including the specialized implementations for horizontal mirror, vertical mirror, and 90-degree rotation. Also, a modified variant of the approach, based on auto-generated machine code is presented. Very high computational performances are achieved at the expense of memory usage. The performances from the perspective of time complexity are analyzed and compared with classical implementation, FPGA implementation, and other implementations of the image rotation. The new method ensures a constant processing time, regardless of the type of transformation. Numerical results are given for a set of different PC specifications to provide full insight into the implementation performances. The processing times for very large images are below 100 ms for most machines, which is 50-60 times faster than the classical implementation, 10-20 times faster than the FPGA implementation, and 2-6 times faster than other implementations of image rotation. Original documents belonging to Nikola Tesla are used for visual demonstration of performance.

Keywords: Linear transformations, Affine transformations, Spatial transformations, Rotation, Machine optimization.

1. Introduction

The performances of OCR systems are usually highly dependent on the input document image which is being processed. Distortions, which can appear in a document image, have to be fixed from the outset using geometrical image transformations. Due to their importance in OCR, such transformations and their implementations have thus long been the subject of research. Although the transformations can be applied individually, generally, their use makes more sense in complex systems (Eldon, 1988; Giulieri, Nolibe, and Richon, 1988; Hagege and Francos, 2005; Liu, D., Yin, Liu, L., Wei, 2013; Schmalz, 1993; Younes, 2006), where they are usually focused on fixing

distortions (Chang and Fitzpatrick, 1990). The emphasis of this new method is linear transformations, but recent research also includes perspective and nonlinear transformations (Devich and Weinhaus, 1980; Evemy, Allerton, and Zaluska, 1989; Weeks, Myler, and Emery, 1994). Geometrical image transformations can be realized using quantum algorithms based on n-qubit normal arbitrary superposition state (NASS) (Fan, Zhou, Jing, and Li, 2016). Handwritten documents can be classified using affine transformation and 2D projection transformation (Yamashita and Wakahara, 2016). A fast FPGA based implementation for 3D affine transform was proposed by Mondal et al. (2016). Evaluation of the linear geometrical image transformations is usually performed by analyzing the implementation time complexity and the interpolation method, which determines the image quality after the transformation is applied. Estimation of the linear transformations by analyzing the periodic properties of interpolation using the second-derivative of the transformed image, was proposed by Ryu and Lee (2014). Affine transformations can be used in order to perform a fractal based image compression (Raittinen and Kaski, 1993). Representation of most classes of 3D objects can be achieved using feed-forward neural networks and affine transformations can be applied to objects in order to prove the possibility of this modeling (Piperakis and Kumazawa, 2001). Affine transformation can also find application in cryptography for the simultaneous encryption-decryption of two images (Kovalchuk, Peleshko, Navytka, and Sviridova, 2011). Some work deals with application of affine transformations for motion compensation (Lopes and Ghanbari, 2002; Nakaya and Harashima, 2002; Rodrigues, da Silva, and de Faria, 2001). Pham and Nakamura (2015) proposed a new algorithm based on affine transformations for deformation of robot trajectories. Affine transformations can also be performed in the frequency domain (Lucchese, 2001). Pei and Hsiao (2015) proposed a method for spatial affine transformations using a fractional shift Fourier transform. An FPGA implementation of affine transformations was proposed by Bensaali et al. (2003). Decomposition of rotation into a sequence of one-dimensional translations, which represents a fast convolution-based interpolation and preserves the high quality of the rotated image, was described by Unser et al. (1995). High precision rotation angle estimation for rotated images was proposed by Qian et al. (2013). A method which exploits the hidden periodicities in the rotated image in the frequency domain using the 2D spectrum was demonstrated by Chen et al. (2014). The distortion problem arising from repeated image rotation applied on the compressed image was addressed by Yi et al. (2015). An algorithm for image rotation and correction based on local feature was proposed by Li and Dan (2013). Image rotation can be also used in video processing, for example, image rotation algorithm for a traffic monitoring system (Tan, Zhang, and Song, 2012). Image scaling has been accomplished using a real-time FPGA-based hardware architecture for implementation of bicubic interpolation (HABI) (Nuno-Maganda and Aries-Estrada, 2005). Another FPGA implementation of image rotation in video was proposed by Berthaud et al. (1998), which exploits B-spline interpolation.

As mentioned previously, geometrical image transformations normally form part of a complex system, such as an OCR algorithm. Geometrical image transformations are applied in a very early stage of the character segmentation process, since further processing is impossible if distortions are present. The most frequent use of geometrical image transformation is for document image skew correction. Document skew correction is applied in combination with document skew estimation, and image rotation is used for skew correction process (Yu and Jain, 1996). To achieve

real-time character recognition of Chinese documents for a reading robot, Yu et al. (2006) proposed a fast image rotation algorithm. Another image rotation algorithm used for document skew correction avoids multiplications in order to reduce the computational cost (Cao, Wang, and Li, 2003). Kapoor et al. (2004) proposed a document skew correction algorithm which exploits the Radon transform. In order to ensure distortion-free rotation, document skew correction can also be achieved using multi-rate signal processing principles (Mahata and Ramakrishnan, 2000).

One design objective of the new method was to achieve very high computational performances, since it is intended for real-time OCR systems. The approach is based on pre-computed lookup tables for mapping offsets to achieve a constant processing time for all transformations, regardless of the complexity of the transformation mapping functions. To provide the highly-optimized implementation, linear image representation is used. Implementations using pointer arithmetic and highly-optimized machine code are used for optimal efficiency. Implementation time complexity is evaluated on several different PC machines, to provide a range of results for a fuller understanding of performance. Results show that the processing times are below 100 ms for most machines, even for very large images. Pointer arithmetic and highly optimized machine code implementations proved to be almost 50 times faster than the classical procedure for image rotation, 10-20 times faster than the FPGA implementation, and 2-6 times faster than diverse algorithms for forward and inverse image rotation. Also, the image rotation quality using the proposed approach is quantitatively compared with equivalent Photoshop® rotation. Beside the optimized implementations of the new method, specialized procedures for horizontal mirror, vertical mirror, and 90-degree rotation are provided along with performance results. Since the new technique requires three memory accesses which limits the processing time, a modified implementation based on auto-generated machine code is also presented. The visual performances of these new procedures are demonstrated using original Nikola Tesla documents from the “Nikola Tesla Museum” in Belgrade. The numerical results demonstrate the new method is ideal for implementation in real-time systems.

This paper is organized as follows: section 2 provides description of the related work with the focus on image rotation, which are used later for comparison and evaluation of the new method. Section 3 provides the detailed mathematical background of linear transformations and gives general description of transformation matrices used for spatial image transformations. In section 4 the complete approach is presented, including the crucial optimization steps which enable fast implementation. Section 5 offers more detail about the fast implementations using pointer arithmetic and highly-optimized low-level machine code. In section 6, experimental results for time complexity are presented using large images with different dimensions. In the concluding section 7, the pros and cons of the new method are discussed, in addition to plans for future work.

2. Related work

Within the field of linear image transformations, image rotation and scaling are frequent research topics. Related work is usually focused on efficient implementations without compromising image quality.

Zhu et al. (2016) proposed a learning-to-rank approach for estimation of the image scaling factor. This uses the normalized energy density features and moment features and is based on training the parameters which represent the difference of previously mentioned features for ordered image pairs.

Ashtari et al. (2015) proposed a fast image rotation algorithm which preserves quality of the image. The method determines base-line equation on the target image and uses floating-point multiplications for this task. Other lines are determined using the base-line pixel coordinates.

Cheng and Wan (2015) proposed an image rotation which uses a radial basis function (RBF). The advantages are related to the improvement of the interpolation mechanism, giving better results than all classical interpolation methods, including the method based on Hermite basis expansion.

The previously described image rotation approaches mainly use image processing techniques in the spatial domain. Fu and Wan (2015) proposed image rotation based in the frequency domain using a discrete cosine transform (DCT). This involves performing a two-dimensional DCT on image blocks for obtaining the frequency domain information, followed by a two-dimensional IDCT before the final interpolation.

In order to evaluate the computational performances of the new method, it is necessary to give more details about the approaches which are used for comparison. Bourennane et al. (2002) proposed a real-time image rotation implementation using FPGAs. For this task, static and dynamic reconfiguration of the FPGA are used and their performances are compared. It is shown that, depending on the nature of application, dynamic reconfiguration gives better results in general. The processing time for the image (by the IIR casual filter) is as follows:

$$T_{IIRCASUAL} = T_{DITER} * \frac{Image\ Size}{Data\ Parallelism\ Ratio} * 3T \quad (1)$$

where T_{DITER} is the iteration period of the dynamic implementation and is equal to 50 ns, the data parallelism ratio is equal to 4, and T represents translation. The overall processing time for image rotation can be calculated as follows:

$$T_{ROTATION} = T_{IIRCASUAL} + T_{IIRANTICASUAL} + T_{FIR} \quad (2)$$

where $T_{IIRANTICASUAL}$ is equal to $T_{IIRCASUAL}$ and T_{FIR} is twice the $T_{IIRCASUAL}$. It should be mentioned that these parameters apply to the specific configuration which is used. The previous equations are used for obtaining results for comparison that are provided in the experimental section of this paper.

Singh et al. (2008) analyzed Hough transform based fast skew detection and accurate skew correction methods. They noticed that forward and inverse image rotation do not give the same results regarding the time complexity performances. Four different image rotation algorithms are

presented, with varying computational complexity, and processing time for each are shown. These results are finally compared with those obtained with the new method.

3. Theoretical background

This section provides a theoretical background for linear transformations and transformation matrices for spatial image transformations, which are the focus of this research.

3.1 Linear transformations

Linear transformations are represented as a mapping from one vector space to another vector space using a mapping function. If V and W are two vector spaces, function T which maps vector space V into vector space W is defined as follows:

$T: V \to W$

where vector space V is also called the domain of T, and vector space W the codomain of T. Functions which map one vector space to another vector space and preserve the operations of vector addition and scalar multiplication are called linear transformations.

Definition 1. Let V and W be vector spaces. The function T: V→W is called a linear transformation of V into W if the following two properties are true for all u and v in V and for any scalar c:

1. $T(u+v) = T(u) + T(v)$
2. $T(cu) = cT(u)$

Theorem 1. Let V and W be two vector spaces and T be a linear transformation from V into W. Then the following properties are true:

1. $T(0) = 0$
2. $T(-v) = -T(v), \forall v \in V$
3. $T(u-v) = T(u) - T(v), \forall u, v \in V$
4. $If\ v = c_1v_1 + c_2v_2 + \cdots + c_nv_n,$

   $then$

   $T(v) = T(c_1v_1 + c_2v_2 + \cdots + c_nv_n) = c_1T(v_1) + c_2T(v_2) + \cdots + c_nT(v_n).$

Proof. Using property (2) of the definition 1, it follows that:

$T(0) = T(00) = 0T(0) = 0.$

Similarly, using property (2) of the definition 1, it follows that:

$T(-v) = T(-1v) = -1T(v) = -T(v).$

Using properties (1) and (2) from the definition 1, it follows that:

$T(u-v) = T(u + (-v)) = T(u + (-1)v) = T(u) + T((-1)v) = T(u) + (-1)T(v)$

$$= T(u) + \big(-T(v)\big) = T(u) - T(v).$$

Property (4) from theorem 1 can be proven using induction, on n. For n = 1, property (2) from the definition 1 implies that:

$T(c_1 v_1) = c_1 T(v_1)$.

For n = 2, using properties (1) and (2) from the definition 1:

$T(c_1 v_1 + c_2 v_2) = T(c_1 v_1) + T(c_2 v_2) = c_1 T(v_1) + c_2 T(v_2)$.

Assume that property (4) from theorem 1 stands for n – 1 vectors, thus:

$$T(c_1 v_1 + c_2 v_2 + \cdots + c_n v_n) = T(c_1 v_1 + c_2 v_2 + \cdots + c_{n-1} v_{n-1}) + T(c_n v_n)$$
$$= \big(c_1 T(v_1) + c_2 T(v_2) + \cdots + c_{n-1} T(v_{n-1})\big) + c_n T(v_n).$$

Ending with the previous equation, the proof is completed.

Theorem 2. Let A be a matrix of size m x n. Given a vector

$$\mathrm{v} = \begin{bmatrix} v_1 \\ v_2 \\ . \\ . \\ . \\ v_n \end{bmatrix} \in \mathbb{R}^{\mathrm{n}} \quad defines \quad T(\mathrm{v}) = A\mathrm{v} = A \begin{bmatrix} v_1 \\ v_2 \\ . \\ . \\ . \\ v_n \end{bmatrix} \in \mathbb{R}^{\mathrm{m}}.$$

Then T(v) is a linear transformation from $\mathbb{R}^{\mathrm{n}}$ into $\mathbb{R}^{\mathrm{m}}$.

Remark: It should be proved that matrix A of size m x n defines a linear transformation from $\mathbb{R}^{\mathrm{n}}$ into $\mathbb{R}^{\mathrm{m}}$.

$$\mathrm{Av} = \begin{bmatrix} a_{11} & a_{12} & . & . & . & a_{1n} \\ a_{21} & a_{22} & . & . & . & a_{2n} \\ . & . & & & & . \\ . & . & & & & . \\ . & . & & & & . \\ a_{m1} & a_{m2} & . & . & . & a_{mn} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ . \\ . \\ . \\ v_n \end{bmatrix} = \begin{bmatrix} a_{11}v_1 + a_{12}v_2 + \cdots + a_{1n}v_n \\ a_{21}v_1 + a_{22}v_2 + \cdots + a_{2n}v_n \\ . \\ . \\ . \\ a_{m1}v_1 + a_{m2}v_2 + \cdots + a_{mn}v_n \end{bmatrix}.$$

Proof. Properties (1) and (2) from the definition 1, for u, v ∈ $\mathbb{R}^{\mathrm{n}}$ and scalar c, imply that:

$T(\mathrm{u} + \mathrm{v}) = A(\mathrm{u} + \mathrm{v}) = A(\mathrm{u}) + A(\mathrm{v}) = T(\mathrm{u}) + T(\mathrm{v})$

and

$T(c\mathrm{u}) = A(c\mathrm{u}) = cA(\mathrm{u}) = cT(\mathrm{u})$.

With previous equation, the proof is completed.

3.2 Transformation matrices

The linear transformation T: V→W represents a form for general vector spaces. In this particular case, linear transformation T: $\mathbb{R}^n \to \mathbb{R}^m$ is taken into consideration. If vectors in $\mathbb{R}^n$ are represented as column matrices, the standard basis of $\mathbb{R}^n$ is as follows:

$$B = \{e_1, e_2, \dots, e_n\} = \left\{ \begin{bmatrix} 1 \\ 0 \\ . \\ . \\ . \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ . \\ . \\ . \\ 0 \end{bmatrix}, \dots, \begin{bmatrix} 0 \\ 0 \\ . \\ . \\ . \\ 1 \end{bmatrix} \right\}$$

Theorem 3. Let T: $\mathbb{R}^n \to \mathbb{R}^m$ be a linear transformation. The following vectors can be written:

$$\mathrm{T}(\mathrm{e}_1) = \begin{bmatrix} a_{11} \\ a_{21} \\ . \\ . \\ . \\ a_{m1} \end{bmatrix}, T(e_2) = \begin{bmatrix} a_{12} \\ a_{22} \\ . \\ . \\ . \\ a_{m2} \end{bmatrix}, \dots, T(e_n) = \begin{bmatrix} a_{1n} \\ a_{2n} \\ . \\ . \\ . \\ a_{mn} \end{bmatrix}.$$

These vectors represent columns of matrix A, size of m x n:

$$A = \begin{bmatrix} a_{11} & a_{12} & . & . & . & a_{1n} \\ a_{21} & a_{22} & . & . & . & a_{2n} \\ . & . & & & & . \\ . & . & & & & . \\ . & . & & & & . \\ a_{m1} & a_{m2} & . & . & . & a_{mn} \end{bmatrix}.$$

The matrix A has the property that:

$$T(\mathrm{v}) = A\mathrm{v}, \forall \mathrm{v} \in \mathbb{R}^n.$$

The matrix A is called the standard matrix of T.

Proof. It can be written:

$$\mathrm{v} = \begin{bmatrix} v_1 \\ v_2 \\ . \\ . \\ . \\ v_n \end{bmatrix} = v_1 e_1 + v_2 e_2 + \cdots + v_n e_n, \mathrm{v} \in \mathbb{R}^n.$$

Also, this implies:

$$A\mathrm{v} = \begin{bmatrix} a_{11} & a_{12} & . & . & . & a_{1n} \\ a_{21} & a_{22} & . & . & . & a_{2n} \\ . & . & & & & . \\ . & . & & & & . \\ . & . & & & & . \\ a_{m1} & a_{m2} & . & . & . & a_{mn} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ . \\ . \\ . \\ v_n \end{bmatrix} = v_1 \begin{bmatrix} a_{11} \\ a_{21} \\ . \\ . \\ . \\ a_{m1} \end{bmatrix} + v_2 \begin{bmatrix} a_{12} \\ a_{22} \\ . \\ . \\ . \\ a_{m2} \end{bmatrix} + \cdots + v_n \begin{bmatrix} a_{1n} \\ a_{2n} \\ . \\ . \\ . \\ a_{mn} \end{bmatrix}$$

$$= v_1 T(e_1) + v_2 T(e_2) + \cdots + v_n T(e_n) = T(\mathrm{v}).$$

The proof is completed.

Spatial image transformation represents the mapping of each coordinate pair (x, y) in the input image to the coordinate pair (x', y') in the output image using the mapping functions. This mapping can be described as follows:

$$[x', y'] = [X(x, y), Y(x, y)] \tag{3}$$

where X and Y are general mapping functions. Since the digital images in the input and output are discrete, x and y coordinates are integer-valued. On the other hand, mapping functions X and Y give real-valued outputs, thus this is the mapping from the set of integer values to the set of real values. In the case of continuous input, such mapping would be straightforward but since the images are discrete, the nature of mapping functions complicate the mapping process, manifested through anomalies called holes and overlaps. Holes appear when none of the pixels from the input are mapped to the given valid position in the output, while overlaps occur when one or more pixels from the input are mapped to the same position in the output.

Geometrical 2D transformations of particular interest in this paper are basic linear transformations. These transformations can be represented using the transformation matrix which is used for calculation of the new coordinates. The pixel at position (x, y) in the input image is mapped to position (x', y') using the general transformation matrix as follows:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{4}$$

where coordinates are represented as homogenous coordinates in order to represent 2D geometrical transformations as matrix multiplication.

A translation transformation is used to move the object from one place to another. The transformation matrix used for 2D translation is as follows:

$$T = \begin{bmatrix} 1 & 0 & T_X \\ 0 & 1 & T_Y \\ 0 & 0 & 1 \end{bmatrix} \tag{5}$$

where T_x and T_y are offsets on the X and Y axis, respectively. The transformation pair is obtained by replacing the general transformation matrix with the matrix T:

$$x' = x + T_X \tag{6}$$

$$y' = y + T_Y \tag{7}$$

Scaling transformation is used to change the dimensions of the object. The transformation matrix used for 2D scaling is as follows:

$$S = \begin{bmatrix} S_X & 0 & 0 \\ 0 & S_Y & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{8}$$

where S_x and S_y are scaling factors. The transformation pair is obtained by replacing the general transformation matrix with the matrix S:

$$x' = x * S_X \tag{9}$$

$$y' = y * S_Y \tag{10}$$

Rotation transformation changes the orientation of the object by rotating the object for the given angle. The transformation matrix used for 2D rotation is as follows:

$$R = \begin{bmatrix} cos\,\theta & -sin\,\theta & 0 \\ sin\,\theta & cos\,\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{11}$$

where θ is the rotation angle. The transformation pair is obtained by replacing the general transformation matrix with the matrix R:

$$x' = xcos(\theta) - ysin(\theta) \tag{12}$$

$$y' = xsin(\theta) + ycos(\theta) \tag{13}$$

A reflection transformation flips the object over a line known as a mirror line. In fact, the mirror line can be in any direction, but usually it is along the x-axis or y-axis. Thus, two different reflection transformation matrices are given in this section. The transformation matrix used for reflection over the x-axis is as follows:

$$F = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{14}$$

The transformation pair is obtained by replacing the general transformation matrix with the matrix F:

$$x' = x \tag{15}$$

$$y' = -y \tag{16}$$

The same applies for reflection transformation over the y-axis. The transformation matrix and transformation pair are as follows:

$$F = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{17}$$

$$x' = -x \tag{18}$$

$$y' = y \tag{19}$$

Shearing transformation changes the shape of the object by allowing one edge to be unchanged, and then moving the other pixels in the same direction by a distance equal to the perpendicular distance of that edge to the pixel which is moved. The transformation matrix used for 2D shearing is as follows:

$$H = \begin{bmatrix} 1 & H_Y & 0 \\ H_X & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{20}$$

where H_x and H_y are shearing factors. The transformation pair is obtained by replacing the general transformation matrix with the matrix H:

$$x' = x + y * H_Y \tag{21}$$

$$y' = y + x * H_X \tag{22}$$

The composite transformation matrix is determined by multiplying all transformation matrices which participate in the composite transformation. The composite transformation matrix is calculated as follows:

$$C = T_1 * T_2 * T_3 \tag{23}$$

where T_1, T_2, and T_3 are transformation matrices for individual transformations. Transformations are applied in the same order as they are multiplied. It should be noted that this is not a commutative operation.

4. General ultra-fast image transformation approach

The ultra-fast approach for geometrical image transformations, which is presented in this section, is based on the usage of lookup tables which contain offsets of pixel positions. In order to perform efficient geometrical transformations, it is necessary to generate mapping offsets for each transformation which will be used. These pre-computed mapping offsets represent the input-output mapping of coordinate positions. The input-output mapping scheme is shown in Fig. 1.

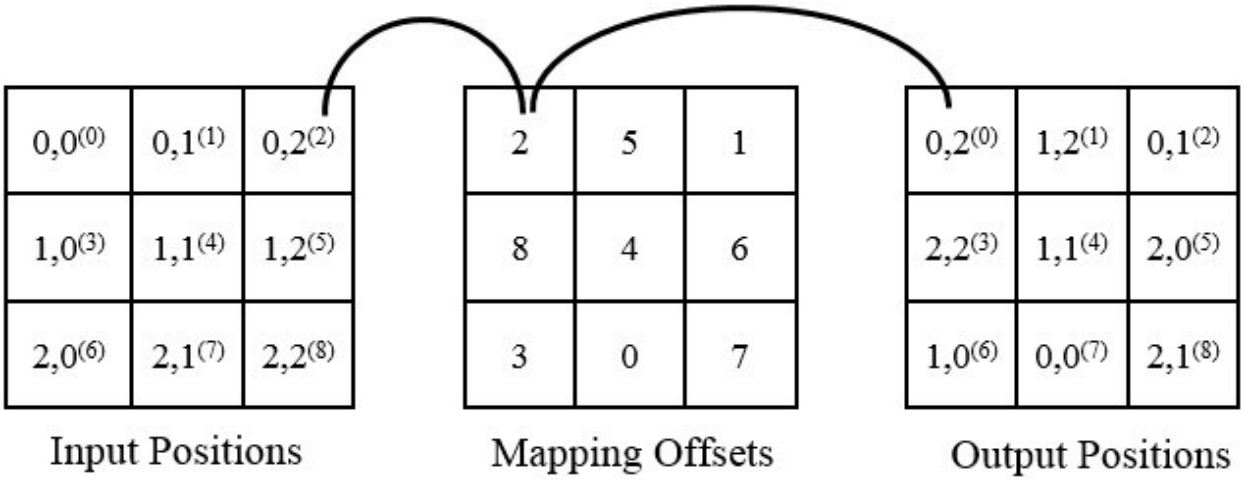

Fig. 1. Mapping scheme.

The mapping scheme shown in Fig. 1 corresponds to the image size of 3x3. The input position represented with coordinates (0,2) and offset 2 is mapped to the offset 0 position, the input position with coordinates (1,2) and offset 5 is mapped to offset 1, and so on. Finally, input position with coordinates (2,1) and offset 7 is mapped to the output with offset 8.

This approach is especially effective when transformations need to be applied multiple times. At the very initial stage it is necessary to compute mapping offsets for transformations using specific parameters. Since it is not possible to compute mapping offsets for all possible transformations, composite transformations can be obtained by calculating transformation mapping offsets from standard transformations.

Using the previously described architecture, it is possible to highly reduce the number of computations, i.e. to minimize the computational cost. In fact, all computations are moved to the

very initial stage, and held in memory, and are recalled later when it is necessary to apply transformations. This ensures the constant processing time for all transformations, which is dependent only on memory access time. This architecture is general and applicable to all other spatial transformations which can be represented using the input-output mapping.

This optimization is achieved at the expense of memory usage, since it is necessary to store all transformation lookup tables in runtime, and use them each time the transformation is required.

In order to achieve optimized implementation, the image is represented as a one-dimensional array of pixel intensity values, the same way it is stored in the memory. This linear image representation is shown in Fig. 2.

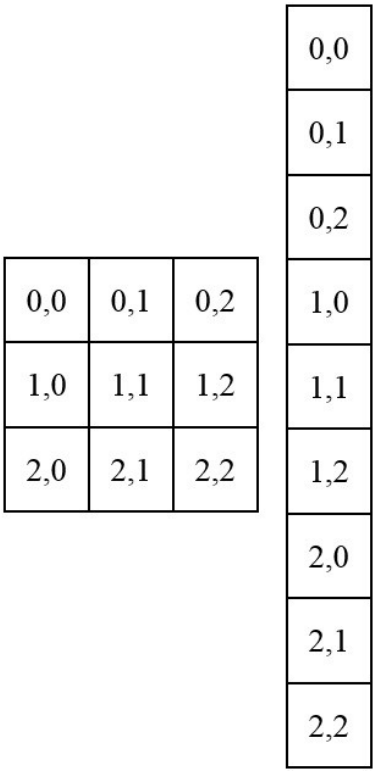

Fig. 2. Linear image representation.

The linear image representation is a crucial step in achieving a highly-optimized implementation. Since the memory has a linear organization, linear image representation ensures direct access to the successive memory locations, which is the most efficient access. An efficient input-output mapping scheme is obtained by representing the standard mapping scheme as a one-dimensional scheme. The linear mapping scheme for fast implementation is shown in Fig. 3.

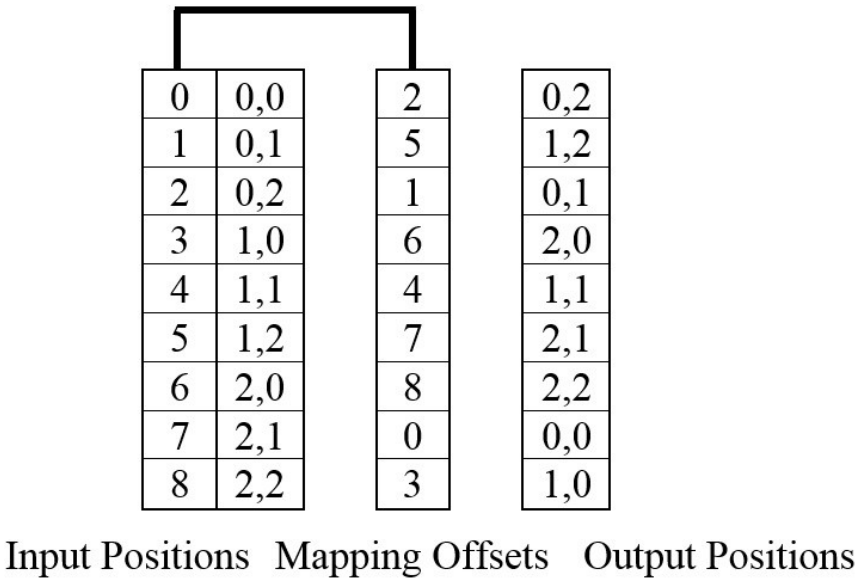

Fig. 3. Linear mapping scheme for fast implementation.

The mapping scheme is visually simpler in this case. The input position with offset 2 is mapped to offset 0, position with offset 5 is mapped to offset 1, and so on. Finally, input position with offset 3 is mapped to the output with offset 8.

Further optimization is achieved by representing the pixel intensity value as a 32-bit integer value. This step ensures a direct access to pixel intensity values. This value is determined as follows:

$$P_{VAL} = R * 256^2 + G * 256^1 + B * 256^0 \quad (24)$$

where R, G, and B are pixel intensity value components.

Beside these optimizations, another important aspect is the usage of other support lookup tables. Employing such tables for complex mapping functions, e.g. values of trigonometric functions, are a standard, simple but effective technique to avoid repeating the same computation multiple times within large loops.

5. Fast implementation

This section provides details of several different geometrical transformations. The first stage of the new approach, i.e. generation of the transformation lookup tables, is achieved using pointer arithmetic. Since this process only need be executed once at beginning of run time, it does not determine the transformation processing time, thus it does not require a highly-optimized implementation. The second stage determines the processing time of the proposed approach, thus a highly-optimized low-level machine code is used here. Whilst effective modern compilers generate acceptable machine code, there is often still potential for further optimization. Such optimization is most effective in large loops, when reducing the number of machine instructions inside a loop could save significant processing time.

In order to test low-level machine code implementations, different approaches are used and will be described in the following subsections.

5.1 Implementation using offsets

The first type of implementation is based on the usage of the pre-computed mapping offsets in order to perform a chosen transformation. This is the standard implementation and completely corresponds to the proposed approach. In order to test all methods which belong to this group, three types of mapping offset based approaches are provided.

Implementation employing 32-bit registers is standard. Thus three different 32-bit machine code implementations are provided in this subsection, including their pointer arithmetic implementation counterparts. The standard pointer arithmetic implementation without any optimization is shown in Listing 1 and its corresponding machine code implementation is shown in Listing 2.

Listing 1. Standard pointer arithmetic implementation (STANDARD1-PTR).

```
for I := 0 to Count - 1 do                {Main loop}
begin
  if RPtr^ <> -1 then                     {If current transformation offset is valid}
  begin
    ImageTempPtr := ImageSrcPtr;          {Get the pointer to the source image}
    Inc(ImageTempPtr, RPtr^);             {Add current transformation offset}
    ImageDstPtr^ := ImageTempPtr^;        {Store value to the destination}
  end
  else
    ImageDstPtr^ := WHITE_COLOR;          {Store white pixel to the destination}

  Inc(ImageDstPtr);                       {Increment pointer}
  Inc(RPtr);                              {Increment pointer}
end;
```

Listing 2. Standard machine code implementation using 32-bit registers (STANDARD1-ASM32).

```
asm
  pushad                                  {Push all registers to stack}
    mov ecx,Count                         {Number of pixels to process}
    mov ebx,RPtr                          {Pointer to transformation array}
    mov esi,ImageSrcPtr                   {Source image pointer}
    mov edi,ImageDstPtr                   {Destination image pointer}
@main:                                    {Main loop}
    mov eax,[ebx]                         {Get transformation offset}
    mov edx,WHITE_COLOR                   {Store white color definition to EDX}
    or eax,eax                            {Is it -1?}
    js @init                              {If true, jump further}
    shl eax,2                             {Offset * 4}
    add eax,esi                           {Final pointer}
    mov edx,[eax]                         {Load pixel value from source}
```

```
@init:                          {Label @init}
   mov [edi],edx                {Store pixel value to destination}
   add edi,4                    {Increment 32-bit pointer}
   add ebx,4                    {Increment 32-bit pointer}
  loop @main                    {Loop again through ECX counter}
 popad                          {Pop up all registers from stack}
end;
```

The machine code implementation shown in Listing 2 represents a basic implementation. Like all implementations in this subsection, it requires three memory accesses. The machine code implementation shown in Listing 3 uses LODSD and STOSD instructions which are used in order to optimize the memory access.

Listing 3. Standard machine code implementation using 32-bit registers (STANDARD2-ASM32).

```
asm
 pushad                         {Push all registers to stack}
 mov ecx,Count                  {Number of pixels to process}
 mov esi,RPtr                   {Pointer to transformation array}
 mov ebx,ImageSrcPtr            {Source image pointer}
 mov edi,ImageDstPtr            {Destination image pointer}
@main:                          {Main loop}
 LODSD                          {Load current offset from transformation array}
 mov edx,eax                    {Save current offset}
 or eax,eax                     {Is it -1?}
 js @init                       {If true, jump to label @init}
 shl edx,2                      {Offset * 4}
 mov eax,[edx+ebx]              {Calculate the final offset and load value to EAX}
 STOSD                          {Store loaded value from EAX to destination}
 dec ecx                        {Decrement counter}
 jnz @main                      {If not zero, loop again through ECX}
 jmp @ex                        {Else, jump to label @ex}
@init:                          {Label @init}
 mov eax,WHITE_COLOR            {Store white color definition to EAX}
 STOSD                          {Store value from EAX to destination}
 dec ecx                        {Decrement counter}
 jnz @main                      {If not zero, loop again through ECX}
@ex:                            {Label @ex}
 popad                          {Pop up all registers from stack}
end;
```

As previously mentioned, the crucial aspect of the proposed approach is the usage of the pre-computed mapping offsets. The processing time is thus completely determined by the number of memory accesses. The machine routine shown in Listing 3 has three memory accesses inside the main loop. In order to optimize this routine still further, more efficient LODSD and STOSD instructions are used for obtaining the mapping offsets from the transformation array. The drawback of this procedure is processing of the mapping offsets with a value equal to -1. Since the

processing of mapping offsets is performed inside a long loop, this conditional statement can take a significant processing time and should be eliminated. The machine routine which uses 32-bit registers is optimized primarily by eliminating the conditional statement and is shown in Listing 4.

Listing 4. Optimized machine code implementation using 32-bit registers (STANDARDOPT-ASM32).

```
asm
  pushad                  {Push all registers to stack}
  mov ecx,Count           {Number of pixels to process}
  shr ecx,1               {Reduce number of loops to Count/2 since two pixels are
                          processed at once}
  mov esi,RPTR            {Pointer to transformation array}
  mov ebx,ImageSrcPtr     {Source image pointer}
  mov edi,ImageDstPtr     {Destination image pointer}
 @main:                   {Main loop}
  LODSD                   {Load current offset from transformation array}
  mov eax,[eax+ebx]       {Load pixel intensity value from calculated offset to EAX}
  STOSD                   {Store the loaded value to destination}
  LODSD                   {Load current offset from transformation array}
  mov eax,[eax+ebx]       {Load pixel intensity value from calculated offset to EAX}
  STOSD                   {Store the loaded value to destination}
  loop @main              {Loop again through ECX}
  popad                   {Pop up all registers from stack}
end;
```

Compared with the procedure shown in Listing 3, Listing 4 has three improvements. The first is that the conditional statement is eliminated. In the previous version of the machine routine, mapping offsets with value -1 are expected (denoting an invalid offset) and thus a conditional statement is necessary to check for such cases. The procedure from Listing 4 treats each offset as valid and thus eliminates the conditional statement. The second improvement is the processing of two pixels in one iteration, since the total number of pixels is always even. Lastly the offset value is already prepared and multiplied by 4 in the process of lookup table generation.

Since the procedures from Listing 3 and Listing 4, which use 32-bit registers, are highly optimized and there is no potential for further optimization, one more implementation which exploits 64-bit registers is performed. This implementation uses the MMX instruction set. MMX was the first set of SIMD extensions applied to Intel's 80x86 instruction set. It uses the instructions which operate on either a single 64-bit quantity or simultaneously on 2 32-bit quantities, 4 16-bit quantities, or 8 8-bit quantities. For this purpose, 8 64-bit general purpose registers are available for use. In this concrete case, these registers are used to process two pixels at once. Listing 5 shows the MMX machine code implementation of the generalized procedure for geometrical image transformations.

Listing 5. Machine code implementation using 64-bit registers (STANDARD-ASM64).

```
asm
  pushad                  {Push all registers to stack}
  mov ecx,Count           {Number of pixels to process}
  shr ecx,1               {Reduce number of loops to Count/2 since two pixels are
                           processed at once}
  mov esi,RPTR            {Pointer to transformation array}
  mov ebx,ImageSrcPtr     {Source image pointer}
  mov edi,ImageDstPtr     {Destination image pointer}
@main1:                   {Main loop}
  movq MM0,[esi]          {Load two 32-bit offsets at once}
  add esi,8               {Increment pointer for next pair of offsets}
  movd edx,MM0            {Get the first 32-bit offset}
  or edx,edx              {Is it -1?}
  js @init1               {If true, skip further}
  shl edx,2               {Offset1 * 4}
  mov ebp,[edx+ebx]       {Store the value of the first pixel to EBP}
@main2:                   {Label @main2}
  psrlq MM0,32            {Logical shift right value in register MM0 by 32 positions in
                           order to get the second 32-bit offset}
  movd edx,MM0            {Get the second 32-bit offset}
  or edx,edx              {Is it -1?}
  js @init2               {If true, skip further}
  shl edx,2               {Offset2 * 4}
  mov eax,[edx+ebx]       {Store the value of the second pixel intensity value to EAX}
@store:                   {Label @store}
  mov [edi],ebp           {Store the value for first pixel to destination}
  mov [edi+4],eax         {Store the value for second pixel to destination}
  add edi,8               {Increment destination pointer}
  dec ecx                 {Decrement counter}
  jnz @main1              {If not zero, loop again though ECX}
  jmp @ex                 {If zero, finish processing}
@init1:                   {Label @init1}
  mov ebp,WHITE_COLOR     {Store white color definition to EBP}
  jmp @main2              {Jump to label @main2}
@init2:                   {Label @init2}
  mov eax,WHITE_COLOR     {Store white pixel definition to EAX}
  jmp @store              {Jump to label @store}
@ex:                      {Label @ex}
  popad                   {Pop up all registers from stack}
  emms                    {Exit the MMX mode (since MMX and FPU registers occupy
                           the same space, it is important to explicitly exit the MMX
                           mode)}
end;
```

Since the machine routines are highly-optimized, there is no waste of machine cycles to additional instructions. As will be confirmed later with the experimental results, it is thus clear that the absolute performance limit, determined only by the memory access speed, has been met. Although the MMX instructions use the 64-bit registers, they also need to access the memory and have the same number of read-write accesses as the 32-bit variant of the machine routine. This is why the 64-bit approach is close to the 32-bit approach in processing time. Also, the simple linear image organization, where the pixel intensity values are represented as 32-bit integer values, speed up the processing since there is no need for pre-processing of the pixel intensity values and memory access is direct. Thus, the 32-bit variant of the machine implementation is in the same range as the 64-bit MMX procedure in terms of processing time, and there is no need to exploit the 64-bit approach, since its implementation is more complex than the 32-bit variant. These details imply that the hybrid implementation structure, consisting of 32-bit pointer arithmetic as a base and accelerated 32-bit code (with 64-bit MMX instructions as an auxiliary resource), is optimal for the new method and also for the wider real-time OCR system for which it was designed.

5.2 Implementation using auto-generated machine code

This subsection presents an alternative for the basic new method. The motivation for this implementation is in reduction of the number of memory accesses, since the number of memory accesses determines the processing time. This implementation uses automatic generation of the machine code based on previously calculated mapping offsets. For each mapping offset one machine instruction is generated. This approach requires only two memory accesses since there is no need to load the mapping offsets from memory. By generating the machine instructions from mapping offsets, the complete loading of mapping offsets from memory is moved to the instruction flow. The drawbacks of this implementation are not negligible. It requires machine code generation for each image and each transformation, which means that, in the case of large images, a large number of instructions will be generated. Since auto-generated method uses the Pascal programming language (Delphi), there is a limit to the size of the file which contains the generated machine instructions. This leads to errors and suggests the described implementation is not flexible enough for practical use. A solution is to separate all machine instructions into different files of similar size. In the concrete case, these files are not used as raw source files, but as compiled files. For this purpose, source code with machine instructions is bound with the corresponding unit, i.e. with the procedure inside the unit which must be executed. This kind of organization is well suited for projects with large source files. In fact, this implementation ensures that machine code is executed as a set of successive machine instructions without loops or conditional statements. Listings 7-9 show an example of the described implementation.

Listing 6. Auto-generated file with machine instructions.

```
//up.pas
procedure ROTATE_UP_HALF; assembler;
asm
mov ebx,16777215                          {WHITE_COLOR}
mov ecx,408; mov eax,ebx; rep stosd       {White (invalid) pixels}
                                          {Other (valid) pixels}
mov eax,[esi+5392]; stosd
mov eax,[esi+12420]; stosd
mov eax,[esi+12420]; stosd
mov eax,[esi+19448]; stosd
mov eax,[esi+26476]; stosd
mov eax,[esi+33504]; stosd
mov eax,[esi+33504]; stosd
mov eax,[esi+40532]; stosd
mov eax,[esi+47560]; stosd
mov eax,[esi+47560]; stosd
mov eax,[esi+54588]; stosd
mov eax,[esi+61616]; stosd
mov eax,[esi+61616]; stosd
mov eax,[esi+68644]; stosd
mov eax,[esi+75672]; stosd
mov eax,[esi+82700]; stosd
…
end;
```

Listing 7. Definition of the first procedure which loads the machine instructions.

```
unit HALF_UP;

interface

procedure ROTATE_UP_HALF; assembler;

implementation


{$I up.pas}


begin
end.
```

Listing 8. Main rotation procedure which calls previously shown Unit (procedure) (AUTOGEN).

```
asm
  pushad                          {Push all registers to stack}
  mov esi,ImageSrcPtr             {Source image pointer}
  mov edi,ImageDstPtr             {Destination image pointer}
  cld                             {Clear direction flag}
  CALL ROTATE_UP_HALF             {HALF_UP_UNIT}
  CALL ROTATE_DOWN_HALF           {HALF_DOWN_UNIT}
  popad                           {Pop up all registers from stack}
end;
```

This example uses the upper and lower files of machine instructions which correspond to processing of the upper and lower part of the image, but the provided listings show only the upper half. Listing 8 shows the Unit defining the procedure which in turn is called from the main procedure shown in Listing 9. The called procedure executes the machine instructions which are shown in Listing 7. This way the project is structurally organized, since the large number of machine instructions is encapsulated in a separate file.

5.3 Implementation without offsets

This subsection is not associated with the new method, but rather presents machine code implementations of 90-degree rotation and horizontal and vertical mirror transformations. These implementations are achieved without using the mapping offsets, to benchmark the time complexity of the standard machine code implementation. Listings 9-15 show the pointer arithmetic implementations and their corresponding machine routines for horizontal mirror, vertical mirror, and 90-degree rotation.

Listing 9. Pointer arithmetic implementation for horizontal mirror (HMIRROR_PTR).

```
Temp := 2 * Width;                      {Incrementing value}

ImageSrcPtr := @Image[0];               {Source image pointer}
ImageDstPrt := @ImageAnalyze[Width - 1]; {Destination image pointer set to the end of
                                         the first scanline}
for I := 0 to Height - 1 do
begin
  for J := 0 to Width - 1 do
  begin
    ImageDstPrt^ := ImageSrcPtr^;       {Store pixel intensity value to destination}
    Inc(ImageSrcPtr);                   {Increment source image pointer}
    Dec(ImageDstPrt);                   {Decrement destination image pointer}
  end;

  Inc(ImageDstPrt, Temp);               {Set destination image pointer to the end of
                                         the next scanline}
end;
```

Listing 10. Machine code implementation for horizontal mirror using MMX instruction set (HMIRROR_ASM64).

```
asm
  pushad                    {Push all registers to stack}
  cld                       {Clear direction flag}
  mov esi,ImageSrcPtr       {Source image pointer}
  mov edi,ImageDstPtr       {Destination image pointer}
  mov edx,dheight           {Move image height to EDX}
  mov ebx,dwidth            {Move image width to EBX}
  movd mm1,ebx              {Prepare counter}
  shl ebx,2                 {Prepare the starting offset}
  add edi,ebx               {Calculate final starting offset}
  sub edi,4                 {Decrement calculated offset}
  mov eax,8                 {Store correction value to EAX}
 @main1:                    {Outer loop}
  movd ecx,mm1              {Prepare counter}
  movd mm0,edi              {Save calculated offset}
 @main2:                    {Inner loop}
  LODSD                     {Load first pixel intensity value to EAX}
  mov [edi],eax             {Store loaded value to destination}
  sub edi,4                 {Decrement offset}
  dec ecx                   {Decrement counter}
  jnz @main2                {If not zero, loop again through inner loop}
  movd edi,mm0              {Get the previously saved offset}
  add edi,ebx               {Calculate new offset}
  dec edx                   {Decrement counter}
  jnz @main1                {If not zero, loop again through outer loop}
  popad                     {Pop up all registers from stack}
  emms                      {Exit the MMX mode (since MMX and FPU
                            registers occupy the same space, it is important to
                            explicitly exit the MMX mode)}
end;
```

Listing 11. Machine code implementation of the horizontal mirror transformation used for comparison with machine routine shown in Listing 10 (HMIRROR_ASM64V2).

```
asm
  pushad                    {Push all registers to stack}
  cld                       {Clear direction flag}
  mov esi,ImageSrcPtr       {Source image pointer}
  mov edi,ImageDstPtr       {Destination image pointer}
  mov edx,dheight           {Move image height to EDX}
  mov ebx,dwidth            {Move image width to EBX}
  movd mm1,ebx              {Prepare counter}
  shl ebx,2                 {Prepare the starting offset}
  add edi,ebx               {Calculate final starting offset}
```

```
  sub edi,4                    {Decrement calculated offset}
  mov eax,8                    {Store correction value to EAX}
@main1:                        {Outer loop}
  movd ecx,mm1                 {Prepare counter}
  movd mm0,edi                 {Save calculated offset}
@main2:                        {Inner loop}
  MOVSD                        {Store pixel value from source to destination}
  sub edi,eax                  {Calculate next offset}
  dec ecx                      {Decrement counter}
  jnz @main2                   {If not zero, loop again through inner loop}
  movd edi,mm0                 {Get the previously saved offset}
  add edi,ebx                  {Calculate new offset}
  dec edx                      {Decrement counter}
  jnz @main1                   {If not zero, loop again through outer loop}
  popad                        {Pop up all registers from stack}
  emms                         {Exit the MMX mode (since MMX and FPU
                                registers occupy the same space, it is important to
                                explicitly exit the MMX mode)}
end;
```

Listing 12. Pointer arithmetic implementation for vertical mirror (VMIRROR_PTR).

```
Temp := 2 * Width;                          {Incrementing value}

ImageSrcPtr := @Image[0];                   {Source image pointer}
ImageDstPrt := @ImageAnalyze[Count - Width]; {Destination image pointer set to the start
                                              of the last scanline}
for I := 0 to Height - 1 do
begin
  for J := 0 to Width - 1 do
  begin
    ImageDstPrt^ := ImageSrcPtr^;           {Store pixel intensity value to destination}
    Inc(ImageSrcPtr);                       {Increment source image pointer}
    Inc(ImageDstPrt);                       {Increment destination image pointer}
  end;

  Dec(ImageDstPrt, Temp);                   {Set destination image pointer to the start
                                             of the previous scanline}
end;
```

Listing 13. Machine code implementation for vertical mirror procedure (VMIRROR_ASM32).

```
asm
  pushad                       {Push all registers to stack}
  cld                          {Clear direction flag}
  mov esi,ImageSrcPtr          {Source image pointer}
  mov edi,ImageDstPtr          {Destination image pointer}
  mov eax,dheight              {Move image height to EAX}
```

```
  dec eax                          {Decrement EAX}
  mov ebx,dwidth                   {Move image width to EBX}
  push ebx                         {Push EBX to stack}
  shl ebx,2                        {Width * 4}
  mul ebx                          {Prepare starting offset}
  add edi,eax                      {Calculate final start position}
  mov eax,edi                      {Store final start position to EAX}
  mov edx,dheight                  {Move image height to EDX}
  pop ebp                          {Pop up value (image width) from stack to EBP}
 @main:                            {Main loop}
  mov ecx,ebp                      {Prepare counter}
  rep movsd                        {Store pixel intensity values to destination}
  sub eax,ebx                      {Prepare offset for next iteration}
  mov edi,eax                      {Move offset to EDI}
  dec edx                          {Decrement counter}
  jnz @main                        {If not zero, loop again}
  popad                            {Pop up all registers from stack}
end;
```

Listing 14. Pointer arithmetic implementation for 90-degree rotation (90DEG-PTR).

```
 Temp := 0;                                {Incrementing value}
 ImageSrcPtr := @Image[0];                 {Source image pointer}

 for I := 0 to Width - 1 do
 begin
  ImageDstPrt := @ImageAnalyze[Temp];      {Destination image pointer set to the start of the
                                            first scanline}
  for J := 0 to Height - 1 do
  begin
   ImageDstPrt^ := ImageSrcPtr^;           {Store pixel intensity value to destination}
   Inc(ImageSrcPtr);                       {Increment source image pointer}
   Inc(ImageDstPrt, Width);                {Set destination image pointer to the start of the
                                            next scanline}
  end;

  Inc(Temp);                               {Prepare offset for the next pixel in the first
                                            scanline}
 end;
```

Listing 15. Machine code implementation for 90-degree rotation using MMX instruction set (90DEG-ASM64).

```
asm
  pushad                           {Push all registers to stack}
  mov esi,ImageSrcPtr              {Source image pointer}
  mov edi,ImageDstPtr              {Destination image pointer}
  mov eax,dwidth                   {Move image width to EAX}
```

```
  mov ecx,eax                 {Set counter to EAX}
  shr ecx,1                   {Reduce number of loops to Width/2 since two
                               pixels are processed at once}
  movd mm0,ecx                {Move counter value to MM0}
  dec eax                     {Decrement EAX}
  mov ebx,dheight             {Move image height to EBX}
  shl ebx,2                   {Height * 4}
  mul ebx                     {Prepare starting offset}
  add edi,eax                 {Calculate final start position}
  mov edx,dheight             {Move image height to EDX}
  mov ebp,edi                 {Move final offset to EBP}
@main1:                       {Outer loop}
  movd ecx,mm0                {Set counter}
@main2:                       {Inner loop}
  lodsd                       {Load first pixel intensity value to EAX}
  mov [edi],eax               {Store loaded value to destination}
  sub edi,ebx                 {Prepare offset for next pixel intensity value}
  lodsd                       {Load second pixel intensity value to EAX}
  mov [edi],eax               {Store loaded value to destination}
  sub edi,ebx                 {Prepare offset for next pixel intensity value}
  loop @main2                 {Loop again though inner loop}
  add ebp,4                   {Increment destination pointer}
  mov edi,ebp                 {Set destination pointer}
  dec edx                     {Decrement counter}
  jnz @main1                  {If not zero, loop again through outer loop}
  popad                       {Pop up all registers from stack}
  emms                        {Exit the MMX mode (since MMX and FPU
                               registers occupy the same space, it is important to
                               explicitly exit the MMX mode)}
end;
```

In particular it is worthwhile highlighting here two possible implementations of the horizontal mirror transformation. The difference between these implementations is shown in Listing 16.

Listing 16. The comparison of the HMIRROR-ASM64 and HMIRROR-ASM64V2 machine routines.

HMIRROR-ASM64	HMIRROR-ASM64V2
LODSD	MOVSD
mov [edi],eax	
sub edi,4	sub edi,eax
dec ecx	dec ecx
jnz @main2	jnz @main2

As shown in the experimental section, the machine routine HMIRROR-ASM64 performs much faster than the machine routine HMIRROR-ASM64V2. The reason for this lies in the MOVSD instruction. This instruction implicitly affects the EDI register which is used in the next instruction, and blocks the pipeline. After the MOVSD instruction is replaced with the first two instructions

on the left side in Listing 16, the processor is able to simultaneously execute the next three instructions which belong to independent instruction pipelines.

6. Experimental results

In this section numerical results are given for the various implementations of the mapping offset method described in section 5. Numerical results focus on time complexity but some results are also included for image rotation quality. Testing is performed on several PC machines with varying hardware specifications in order to provide a broad range of results for execution time. Implementations of the new method which use the mapping offsets are used for comparison with image rotation algorithms, but since the processing time is not dependent on the type of transformation, provided processing time for the proposed approach implementations will be the same for all other transformations. These results are compared with FPGA implementation of the image rotation (Bourennane et al., 2002). Also, results obtained using the machine routines are compared with image rotation algorithms (Singh et al., 2008). Results for the proposed approach implementations performed on images of different dimensions and executed on different PC machines are shown in Tables 1-8. The two values for each implementation correspond to the best (left half) and average (right half) processing time.

Table 1. Comparison of the processing time for different implementations of image rotation (AMD Athlon™ X4 840 Quad Core Processor 3.1 GHz).

Image Dimensions (px)	Processing Time (ms)											
	Classical Implementation	Bourennane et al. (2002)	Proposed approach									
			STANDARD1-PTR		STANDARD1-ASM32		STANDARD2-ASM32		STANDARDOPT-ASM32		STANDARD-ASM64	
1000x1000	180.229	37.500	3.944	5.418	3.147	4.485	3.772	5.479	4.107	5.378	3.789	5.340
1500x1500	393.473	84.375	9.362	11.688	6.507	9.419	7.544	11.445	7.766	9.789	7.186	10.492
2000x2000	677.833	150.000	17.626	21.852	13.333	17.822	15.526	21.500	14.441	18.693	12.962	18.451
2500x2500	1049.029	234.375	28.518	31.753	20.433	25.208	25.694	31.561	23.538	26.602	24.162	28.788
3000x3000	1503.133	337.500	38.562	41.872	31.251	33.688	33.196	40.563	37.989	41.823	35.774	41.622
3500x3500	2051.882	459.375	50.937	56.103	40.407	43.641	44.219	54.772	44.667	47.431	41.934	48.687
4000x4000	2679.897	600.000	77.476	84.817	59.883	63.359	69.672	81.170	60.663	64.227	59.214	68.521
4500x4500	3400.561	759.375	95.556	102.534	73.989	79.552	85.492	101.450	82.499	88.438	76.486	87.381
5000x5000	4193.635	937.500	112.565	122.437	92.231	97.183	104.117	119.411	111.362	115.276	102.778	117.768

Table 2. Comparison of the processing time for different implementations of image rotation (AMD Athlon™ 64 X2 Dual Core Processor 5200+ 2.6 GHz).

Image Dimensions (px)	Processing Time (ms)											
	Bourennane et al. (2002)	Proposed approach										
		STANDARD1-PTR		STANDARD1-ASM32		STANDARD2-ASM32		STANDARDOPT-ASM32		STANDARD-ASM64		
1000x1000	37.500	10.368	27.104	7.793	18.717	7.615	19.455	7.124	16.395	9.146	23.823	
1500x1500	84.375	23.086	49.295	17.099	38.575	16.802	31.633	17.068	33.766	20.277	44.519	
2000x2000	150.000	40.775	73.879	28.527	56.179	31.859	54.993	27.236	54.221	39.727	66.779	
2500x2500	234.375	63.475	105.339	42.984	75.516	41.638	73.679	40.466	72.203	65.941	92.270	
3000x3000	337.500	111.197	141.341	48.308	83.112	47.914	82.764	55.041	81.332	68.827	111.064	
3500x3500	459.375	140.539	176.895	98.862	126.581	106.394	132.740	98.111	129.037	127.244	153.868	
4000x4000	600.000	201.350	229.632	101.082	154.772	129.470	159.887	126.145	159.476	165.112	195.569	
4500x4500	759.375	246.992	285.973	131.287	195.538	161.315	201.461	160.791	196.824	196.064	227.616	

5000x5000	937.500	298.749	337.479	197.710	244.943	225.953	258.218	217.075	243.669	307.564	343.253

Table 3. Comparison of the processing time for different implementations of image rotation (Intel® Core™ i3-4150 CPU @ 3.50GHz).

Image Dimensions (px)	Processing Time (ms)										
	Bouren nane et al. (2002)	Proposed approach									
		STANDARD1-PTR		STANDARD1-ASM32		STANDARD2-ASM32		STANDARDOPT-ASM32		STANDARD-ASM64	
1000x1000	37.500	1.872	2.125	2.662	2.952	1.831	1.987	1.894	2.049	2.042	2.159
1500x1500	84.375	4.245	4.735	5.939	6.554	3.419	3.845	3.789	4.235	3.537	3.941
2000x2000	150.000	7.619	8.268	10.746	11.773	6.252	7.168	6.722	7.599	6.076	6.845
2500x2500	234.375	11.639	12.586	16.786	17.648	9.209	10.079	10.606	11.401	9.413	10.234
3000x3000	337.500	16.789	17.843	24.389	25.327	13.596	14.828	15.157	16.377	13.795	14.875
3500x3500	459.375	24.163	25.046	33.278	34.454	18.742	19.948	21.156	22.339	18.975	20.404
4000x4000	600.000	31.321	32.072	43.981	44.727	25.372	26.279	28.002	28.993	25.109	25.894
4500x4500	759.375	41.069	41.730	56.347	57.126	32.584	33.424	35.772	36.944	32.576	33.548
5000x5000	937.500	51.234	53.256	69.096	71.043	41.109	42.381	44.623	45.992	41.031	42.352

Table 4. Comparison of the processing time for different implementations of image rotation (Intel® Core™ i5-750 CPU @ 2.67GHz).

Image Dimensions (px)	Processing Time (ms)										
	Bouren nane et al. (2002)	Proposed approach									
		STANDARD1-PTR		STANDARD1-ASM32		STANDARD2-ASM32		STANDARDOPT-ASM32		STANDARD-ASM64	
1000x1000	37.500	3.179	3.719	3.018	3.628	2.492	3.196	2.647	3.314	2.572	3.340
1500x1500	84.375	7.714	9.959	7.164	8.705	6.399	8.119	6.573	8.375	6.178	9.144
2000x2000	150.000	14.021	15.656	12.615	16.579	11.345	15.113	11.156	16.339	11.061	15.332
2500x2500	234.375	22.523	23.288	20.836	22.491	18.100	20.775	17.099	19.867	17.997	20.889
3000x3000	337.500	32.834	35.382	29.605	31.305	26.810	28.868	27.005	28.397	25.807	27.837
3500x3500	459.375	44.633	45.153	40.696	44.637	36.919	39.797	35.932	37.319	36.243	39.042
4000x4000	600.000	58.139	59.244	51.948	52.478	46.684	47.291	48.279	49.007	45.915	46.511
4500x4500	759.375	74.226	74.873	67.135	67.736	62.016	62.603	60.868	61.679	60.963	61.828
5000x5000	937.500	91.847	92.509	81.883	82.614	72.539	73.254	71.415	72.276	70.939	71.701

Table 5. Comparison of the processing time for different implementations of image rotation (Intel® Core™ i7-920 CPU @ 2.67GHz).

Image Dimensions (px)	Processing Time (ms)										
	Bouren nane et al. (2002)	Proposed approach									
		STANDARD1-PTR		STANDARD1-ASM32		STANDARD2-ASM32		STANDARDOPT-ASM32		STANDARD-ASM64	
1000x1000	37.500	2.969	3.061	2.796	2.884	2.011	2.285	2.231	2.381	2.059	2.238
1500x1500	84.375	6.946	7.126	6.312	6.511	4.684	5.264	4.868	5.193	4.542	4.954
2000x2000	150.000	12.347	12.628	11.155	11.476	8.436	9.044	8.976	9.548	8.151	8.781
2500x2500	234.375	19.405	19.801	17.585	18.039	13.464	14.246	13.658	14.367	12.944	13.751
3000x3000	337.500	28.788	28.961	25.519	25.872	20.612	21.600	20.402	21.295	19.651	20.755
3500x3500	459.375	40.216	40.396	34.940	35.098	28.678	28.824	27.604	27.887	27.415	27.822
4000x4000	600.000	52.834	53.107	45.773	46.009	38.699	38.959	37.569	37.816	37.502	37.739
4500x4500	759.375	66.942	67.342	58.489	58.811	49.769	50.058	46.175	46.456	48.266	48.573
5000x5000	937.500	84.228	84.675	73.182	73.603	62.081	62.468	59.752	60.123	60.496	60.852

Table 6. Comparison of the processing time for different implementations of image rotation (Intel® Core™ i7-4700 MQ Processor 2.4 GHz).

Image Dimensions (px)	Processing Time (ms)										
	Bouren nane et al. (2002)	Proposed approach									
		STANDARD1-PTR		STANDARD1-ASM32		STANDARD2-ASM32		STANDARDOPT-ASM32		STANDARD-ASM64	
1000x1000	37.500	2.554	2.859	3.822	4.106	2.235	2.614	2.385	2.747	2.274	2.647
1500x1500	84.375	5.799	6.239	8.538	9.053	4.265	4.784	5.300	6.301	4.385	5.160
2000x2000	150.000	10.273	10.842	15.5028	15.877	7.584	8.489	9.286	9.923	7.527	8.217
2500x2500	234.375	16.802	20.036	24.038	25.152	12.448	16.594	14.988	18.081	12.557	16.689
3000x3000	337.500	23.665	30.623	35.338	39.606	17.842	20.364	21.757	25.509	18.122	23.822
3500x3500	459.375	32.057	33.219	47.123	54.279	23.295	24.347	28.314	29.242	23.369	24.398
4000x4000	600.000	42.994	43.841	61.623	62.937	31.727	33.109	36.791	39.816	33.360	35.486
4500x4500	759.375	53.666	61.056	78.595	80.449	39.609	42.641	47.615	49.574	39.589	41.848
5000x5000	937.500	67.973	70.348	96.851	103.974	48.901	54.527	58.482	62.085	49.568	54.719

Table 7. Comparison of the processing time for different implementations of image rotation (Intel® Core™2 Quad Q9550 CPU @ 2.83GHz).

Image Dimensions (px)	Processing Time (ms)										
	Bouren nane et al. (2002)	Proposed approach									
		STANDARD1-PTR		STANDARD1-ASM32		STANDARD2-ASM32		STANDARDOPT-ASM32		STANDARD-ASM64	
1000x1000	37.500	3.249	4.116	3.084	3.876	2.645	3.539	2.614	3.596	2.671	3.490
1500x1500	84.375	7.966	9.254	7.252	9.136	6.328	8.338	6.241	8.433	6.191	8.136
2000x2000	150.000	14.313	17.254	13.027	15.735	11.613	14.071	11.217	14.063	11.029	13.836
2500x2500	234.375	23.409	27.506	21.408	25.368	17.865	22.798	17.864	22.684	17.342	22.200
3000x3000	337.500	33.865	36.248	34.433	37.055	27.716	32.024	28.106	35.916	27.044	33.401
3500x3500	459.375	45.845	48.176	42.629	46.283	36.736	40.313	36.259	38.524	35.035	37.679
4000x4000	600.000	59.843	62.378	55.729	58.334	47.651	50.070	47.136	49.709	46.289	49.155
4500x4500	759.375	75.607	79.181	71.240	73.986	59.969	62.985	59.776	62.784	56.755	59.651
5000x5000	937.500	93.867	97.181	88.697	91.944	74.103	77.568	73.869	77.397	72.376	75.948

Table 8. Comparison of the processing time for different implementations of image rotation (Intel® Core™ i5-3470 CPU @ 3.20GHz).

Image Dimensions (px)	Processing Time (ms)										
	Bouren nane et al. (2002)	Proposed approach									
		STANDARD1-PTR		STANDARD1-ASM32		STANDARD2-ASM32		STANDARDOPT-ASM32		STANDARD-ASM64	
1000x1000	37.500	1.933	2.229	2.511	2.979	1.530	1.809	1.749	2.002	1.794	2.097
1500x1500	84.375	4.911	5.439	5.920	6.645	3.803	4.317	4.191	4.669	3.973	4.449
2000x2000	150.000	8.249	8.875	10.928	11.635	6.512	7.136	6.912	7.541	6.602	7.240
2500x2500	234.375	12.206	13.048	16.711	17.755	10.823	12.835	10.940	11.751	11.203	12.762
3000x3000	337.500	19.695	22.693	25.463	27.299	16.039	18.653	15.686	16.835	17.016	18.718
3500x3500	459.375	26.975	27.579	33.932	34.718	21.895	22.951	22.074	23.199	21.398	22.632
4000x4000	600.000	36.222	37.122	44.481	45.543	30.362	31.083	30.746	31.291	31.152	31.675
4500x4500	759.375	44.848	46.692	56.954	57.997	38.631	39.278	38.967	39.652	38.065	38.755
5000x5000	937.500	61.967	65.666	74.893	79.630	52.306	53.388	49.438	50.047	51.802	54.460

The results provided in the previous tables show that all presented implementations are executed in less than 100 ms on the majority PC machines tested. Machine code optimization is the crucial step in optimization of the transformation procedure and the results show that it provides slightly better results than the implementation using pointer arithmetic. The new implementations perform 50-60 times faster than the classical implementation of the image rotation where all computations are performed each time the transformation is applied, and compared with the FPGA implementation of the image rotation (Bourennane et al., 2002), the new implementations perform 10-20 times faster. Results provided in Tables 1-8 show that in most cases processing times are stable, i.e. the difference between the best and average processing time is not significant. In case of older processors, such as the AMD Athlon™ 64 X2 Dual Core Processor 5200+, processing time proved to be unstable and the difference between the best and average processing time is longer. This comes from many factors which can affect the processing speed such as the amount of RAM, CPU generation and speed, bus type and speed, amount of cache memory.

In order to better evaluate the efficiency of the new method and its various implementations, results which correspond to the algorithms for forward and inverse rotation presented by Singh et al. (2008) are shown in Table 9. These results were obtained on a PC machine with 256 MB RAM installed and CPU running at 2.8 GHz.

Table 9. Processing time for different implementations of image rotation (Singh et al. (2008)).

Image dimensions (px)	Processing Time (ms)							
	Singh et al. (2008)							
	Float rotation		Integer rotation		Fast implementation		Bresenham's line like algorithm	
1249x1249	15	172	15	78	15	78	16	31
4148x4068	218	1875	156	844	157	841	235	313

Singh et al. (2008) provided results for forward and inverse rotation, since they noticed that forward and inverse rotation do not give the same results. Taking this into account, it should be mentioned that the new method's processing time is not dependent on the type of the image rotation, since all calculations are performed at the start. Compared with results shown in Table 9, the new implementations give better results in general.

Tables 10-17 show processing time for 90-degree rotation implemented using pointer arithmetic and machine code. These implementations do not use the mapping offsets. Execution time proved to be worse than for implementations of the new method, but still significantly better than results for implementations presented by Bourennane et al. (2002) and Singh et al. (2008).

Table 10. Processing time of pointer arithmetic and machine code implementations for 90-degree rotation (AMD Athlon™ X4 840 Quad Core Processor 3.1 GHz).

Image Dimensions (px)	Processing Time (ms)			
	90DEG-PTR		90DEG-ASM64	
1000x1000	3.383	4.543	3.456	4.624
1500x1500	13.468	16.272	13.305	16.252
2000x2000	28.065	30.365	28.004	30.377
2500x2500	40.358	43.810	38.997	42.700
3000x3000	59.703	63.909	59.059	63.018
3500x3500	77.676	86.883	78.136	87.920
4000x4000	110.349	115.699	108.289	116.714
4500x4500	130.972	143.416	130.520	138.462
5000x5000	168.505	175.755	169.922	177.679

Table 11. Processing time of pointer arithmetic and machine code implementations for 90-degree rotation (AMD Athlon™ 64 X2 Dual Core Processor 5200+ 2.6 GHz).

Image Dimensions (px)	Processing Time (ms)			
	90DEG-PTR		90DEG-ASM64	
1000x1000	13.639	28.590	14.456	32.198
1500x1500	39.839	72.587	42.143	72.856
2000x2000	93.324	121.909	76.102	120.580
2500x2500	145.320	175.554	144.035	175.443
3000x3000	205.769	247.832	208.556	246.887
3500x3500	295.149	331.437	297.415	329.228
4000x4000	424.042	469.191	427.783	468.479
4500x4500	493.559	543.061	456.163	535.224
5000x5000	628.936	672.401	619.644	665.687

Table 12. Processing time of pointer arithmetic and machine code implementations for 90-degree rotation (Intel® Core™ i3-4150 CPU @ 3.50GHz).

Image Dimensions (px)	Processing Time (ms)			
	90DEG-PTR		90DEG-ASM64	
1000x1000	2.288	2.536	2.255	2.531
1500x1500	16.269	17.095	16.232	16.969
2000x2000	30.048	30.996	29.698	30.869
2500x2500	47.926	49.007	47.605	48.655
3000x3000	71.335	74.236	71.852	73.192
3500x3500	113.769	115.280	110.882	115.459
4000x4000	148.840	150.281	147.599	149.203
4500x4500	204.823	208.261	209.891	211.665
5000x5000	255.742	258.427	264.616	266.684

Table 13. Processing time of pointer arithmetic and machine code implementations for 90-degree rotation (Intel® Core™ i5-750 CPU @ 2.67GHz).

Image Dimensions (px)	Processing Time (ms)			
	90DEG-PTR		90DEG-ASM64	
1000x1000	8.947	9.420	8.982	9.425
1500x1500	21.446	22.033	21.581	22.274
2000x2000	39.833	41.337	40.063	41.408
2500x2500	62.113	63.174	62.562	64.199
3000x3000	90.107	91.328	90.516	91.559
3500x3500	130.396	130.915	131.246	131.808
4000x4000	176.872	177.829	178.158	178.709
4500x4500	233.900	234.430	235.472	236.081
5000x5000	296.569	297.385	298.113	298.773

Table 14. Processing time of pointer arithmetic and machine code implementations for 90-degree rotation (Intel® Core™ i7-920 CPU @ 2.67GHz).

Image Dimensions (px)	Processing Time (ms)			
	90DEG-PTR		90DEG-ASM64	
1000x1000	8.917	8.962	8.939	8.985
1500x1500	21.362	21.436	21.512	21.588
2000x2000	37.459	37.613	37.682	37.790
2500x2500	59.060	59.208	59.514	59.703
3000x3000	89.405	89.544	89.616	89.782
3500x3500	132.581	132.694	133.471	133.569
4000x4000	169.524	169.796	170.979	171.089
4500x4500	234.038	234.299	235.792	236.062
5000x5000	295.383	295.664	296.909	297.074

Table 15. Processing time of pointer arithmetic and machine code implementations for 90-degree rotation (Intel® Core™ i7-4700 MQ Processor 2.4 GHz).

Image Dimensions (px)	Processing Time (ms)			
	90DEG-PTR		90DEG-ASM64	
1000x1000	2.814	3.228	2.761	3.154
1500x1500	23.778	24.133	23.648	23.959
2000x2000	42.676	43.585	42.567	43.284
2500x2500	68.182	73.057	67.374	73.212
3000x3000	103.709	115.844	101.988	114.978
3500x3500	160.747	166.040	162.071	165.964
4000x4000	210.756	214.221	209.771	212.417

4500x4500	292.907	301.229	298.777	304.808
5000x5000	365.081	371.070	377.759	385.382

Table 16. Processing time of pointer arithmetic and machine code implementations for 90-degree rotation (Intel® Core™2 Quad Q9550 CPU @ 2.83GHz).

Image Dimensions (px)	Processing Time (ms)			
	90DEG-PTR		90DEG-ASM64	
1000x1000	6.426	7.018	7.091	7.678
1500x1500	14.804	16.010	16.111	17.309
2000x2000	28.612	30.097	31.322	32.537
2500x2500	44.876	47.306	48.919	51.165
3000x3000	69.447	73.631	76.135	79.315
3500x3500	91.814	95.462	100.828	104.381
4000x4000	137.009	141.549	150.954	156.199
4500x4500	162.189	167.991	176.067	182.666
5000x5000	216.765	224.811	231.346	241.429

Table 17. Processing time of pointer arithmetic and machine code implementations for 90-degree rotation (Intel® Core™ i5-3470 CPU @ 3.20GHz).

Image Dimensions (px)	Processing Time (ms)			
	90DEG-PTR		90DEG-ASM64	
1000x1000	6.455	6.886	6.433	6.869
1500x1500	15.388	16.229	15.326	16.135
2000x2000	29.622	30.166	29.396	29.993
2500x2500	45.503	46.372	44.957	45.983
3000x3000	68.446	69.572	67.549	68.708
3500x3500	103.314	105.119	102.793	104.594
4000x4000	138.472	139.850	137.725	139.116
4500x4500	186.031	188.108	187.566	189.515
5000x5000	230.722	232.973	235.461	237.783

Fig. 4. shows comparison of the general processing time for the optimized low-level machine code implementation of the proposed approach, and processing time for classical implementation and FPGA implementation of the image rotation.

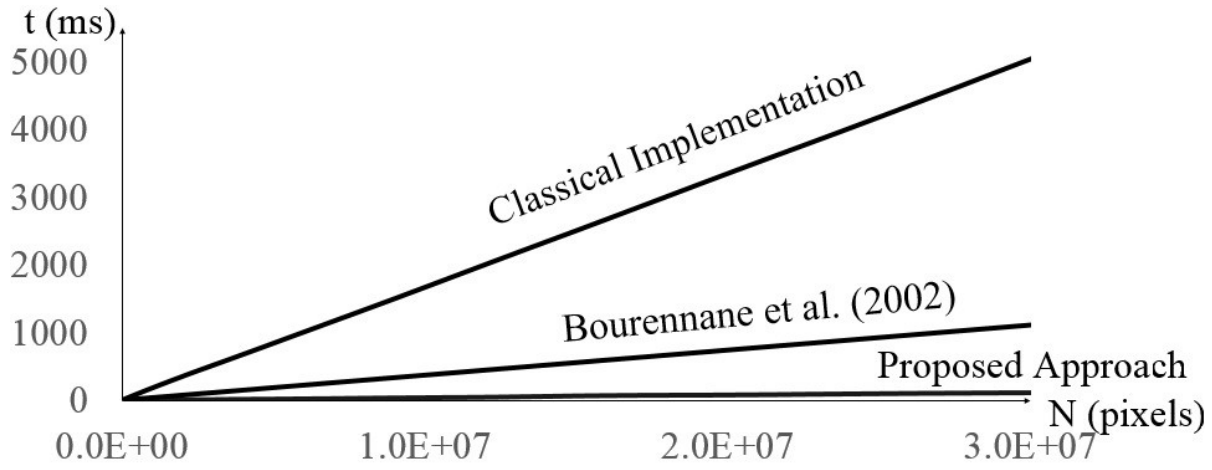

Fig. 4. Comparison of processing time for different image rotation approaches.

As already mentioned, high computational performances are achieved at the expense of the memory usage. Besides faster image transformations, the new method ensures constant processing time for all transformations, thus the processing time is not dependent on the type of transformation. The memory usage is dependent on both the number of transformation lookup tables, which are computed at the outset, as well as on the number of pixels. Therefore, the more pixels to be processed, the fewer transformation lookup tables that can be stored at runtime.

In order to estimate the quality of the image rotation, a random image is rotated by different angles and compared with the equivalent rotation performed in Photoshop®. The peak signal-to-noise-ratio (PSNR), absolute deviation, and standard deviation are metrics used to estimate the quality of rotation. These results are shown in Table 18.

Table 18. PSNR for random image rotated for different angles and compared to the equivalent Photoshop® rotation.

Angle (°)	PSNR (dB)	D (dB)
0.15	26.44	0.28
0.24	27.60	0.88
0.61	27.54	0.82
0.98	28.09	1.37
1.6	23.48	3.24
2.5	27.65	0.93
3.9	24.27	2.45
6.3	28.64	1.92
10	26.74	0.02

$$\sigma = \sqrt{\frac{\sum|x - \bar{x}|^2}{n}} = 1.64836\ [dB] \tag{25}$$

The proposed approach gives satisfactory results with respect to image rotation quality. It should be mentioned that the new implementations adapted for image rotation use nearest neighbor interpolation, since this interpolation method is most suitable.

Further comparisons can be made using pointer arithmetic and machine code implementations of the horizontal and vertical mirror transformations. Processing time for these implementations are given in Tables 19-26.

Table 19. Processing time for pointer arithmetic and machine code implementations for horizontal and vertical mirror transformations (AMD Athlon™ X4 840 Quad Core Processor 3.1 GHz).

Image Dimensions (px)	Processing Time (ms)									
	HMIRROR-PTR		HMIRROR-ASM64		HMIRROR-ASM64V2		VMIRROR-PTR		VMIRROR-ASM32	
1000x1000	1.835	2.699	1.768	2.758	4.171	5.728	1.798	2.653	1.551	2.359
1500x1500	4.285	6.035	4.603	6.428	9.252	13.005	4.006	5.816	3.869	5.465
2000x2000	6.819	10.548	8.887	11.389	15.844	20.981	8.128	10.709	7.774	9.828
2500x2500	10.789	16.622	11.833	17.059	27.860	29.760	11.447	16.309	12.599	15.331
3000x3000	16.373	22.579	16.789	22.928	38.976	41.586	16.205	22.205	16.529	21.679
3500x3500	25.738	32.625	26.243	30.096	51.367	53.697	25.084	31.385	21.739	28.984
4000x4000	32.532	37.615	35.022	39.187	66.256	72.725	32.743	35.566	29.187	35.715
4500x4500	39.953	43.681	43.347	48.439	88.200	93.929	39.935	44.187	39.369	43.997
5000x5000	49.542	55.297	53.396	59.786	107.032	114.016	48.901	54.996	48.711	53.472

Table 20. Processing time for pointer arithmetic and machine code implementations for horizontal and vertical mirror transformations (AMD Athlon™ 64 X2 Dual Core Processor 5200+ 2.6 GHz).

Image Dimensions (px)	Processing Time (ms)									
	HMIRROR-PTR		HMIRROR-ASM64		HMIRROR-ASM64V2		VMIRROR-PTR		VMIRROR-ASM32	
1000x1000	2.919	6.686	2.989	9.275	3.063	9.779	2.579	8.111	2.527	7.779
1500x1500	5.893	17.501	6.069	16.037	6.312	18.503	5.859	17.162	5.851	15.646
2000x2000	10.158	23.965	10.231	27.765	10.557	27.961	10.402	26.498	10.243	27.053
2500x2500	17.301	38.632	17.951	36.953	18.511	38.075	16.746	33.650	16.063	35.286
3000x3000	23.875	50.090	24.578	50.747	25.370	52.559	25.378	49.418	22.897	48.537
3500x3500	32.949	62.542	34.665	64.808	35.843	66.889	32.656	62.388	31.381	60.216
4000x4000	56.659	81.795	49.901	85.462	51.427	88.451	47.055	82.269	59.290	80.964
4500x4500	71.884	98.839	77.837	101.946	63.275	105.497	74.697	98.987	56.560	94.865
5000x5000	64.372	105.554	82.798	107.408	89.377	112.557	81.232	106.743	80.935	104.812

Table 21. Processing time for pointer arithmetic and machine code implementations for horizontal and vertical mirror transformations (Intel® Core™ i3-4150 CPU @ 3.50GHz).

Image Dimensions (px)	Processing Time (ms)									
	HMIRROR-PTR		HMIRROR-ASM64		HMIRROR-ASM64V2		VMIRROR-PTR		VMIRROR-ASM32	
1000x1000	0.594	0.692	0.595	0.709	2.013	2.163	0.599	0.705	0.432	0.523
1500x1500	1.369	1.618	1.367	1.592	4.526	4.825	1.371	1.617	1.092	1.278
2000x2000	2.415	2.732	2.415	2.807	8.043	8.483	2.398	2.787	1.932	2.253
2500x2500	3.753	4.322	3.751	4.322	12.616	13.195	3.783	4.352	3.043	3.535
3000x3000	5.407	6.208	5.392	6.222	18.215	19.051	5.422	6.283	4.392	5.023
3500x3500	7.337	8.389	7.323	8.436	25.101	25.633	7.367	8.421	5.939	6.849
4000x4000	9.668	10.862	9.613	10.864	32.743	33.392	9.674	10.905	7.783	8.851
4500x4500	12.218	13.697	12.182	13.658	41.091	42.080	12.244	13.687	9.943	11.159
5000x5000	15.055	16.687	15.022	16.676	50.731	52.128	15.126	16.718	12.201	13.676

Table 22. Processing time for pointer arithmetic and machine code implementations for horizontal and vertical mirror transformations (Intel® Core™ i5-750 CPU @ 2.67GHz).

Image Dimensions (px)	Processing Time (ms)									
	HMIRROR-PTR		HMIRROR-ASM64		HMIRROR-ASM64V2		VMIRROR-PTR		VMIRROR-ASM32	
1000x1000	1.078	1.544	1.078	1.707	1.792	2.351	1.078	1.545	0.364	0.838
1500x1500	3.719	5.965	3.691	5.139	4.478	6.184	3.811	5.451	2.551	4.354
2000x2000	6.809	10.584	6.608	10.045	7.887	9.646	6.833	10.096	5.047	7.686
2500x2500	11.479	15.693	11.855	14.646	13.026	15.549	11.403	14.684	8.082	11.579
3000x3000	15.090	21.153	15.368	20.355	18.309	23.718	15.779	21.617	11.796	16.986
3500x3500	22.548	24.235	22.904	26.456	26.747	27.854	22.945	24.249	16.001	19.622
4000x4000	25.725	29.839	28.932	29.798	33.899	34.346	27.902	28.934	19.725	22.778
4500x4500	37.969	38.878	37.844	38.815	43.856	44.372	36.952	37.837	28.819	29.797
5000x5000	45.928	47.062	45.576	46.963	52.481	53.316	43.243	44.266	34.736	35.379

Table 23. Processing time for pointer arithmetic and machine code implementations for horizontal and vertical mirror transformations (Intel® Core™ i7-920 CPU @ 2.67GHz).

Image Dimensions (px)	Processing Time (ms)									
	HMIRROR-PTR		HMIRROR-ASM64		HMIRROR-ASM64V2		VMIRROR-PTR		VMIRROR-ASM32	
1000x1000	1.086	1.201	1.085	1.204	1.801	1.872	1.084	1.207	0.363	0.516
1500x1500	2.824	3.112	2.834	3.111	4.173	4.294	2.806	3.109	1.907	2.157
2000x2000	5.031	5.484	5.000	5.479	7.420	7.622	5.082	5.547	3.393	3.864
2500x2500	7.312	8.551	7.874	8.553	11.616	11.908	7.944	8.603	5.307	5.972
3000x3000	11.402	12.205	11.418	12.209	16.677	17.045	11.491	12.385	7.701	8.629
3500x3500	15.715	16.806	15.713	16.737	22.775	23.207	15.623	16.648	10.308	11.295
4000x4000	19.945	21.599	20.285	21.396	29.908	30.079	20.550	21.683	13.981	15.317
4500x4500	24.493	26.762	26.039	26.755	37.751	37.961	26.424	26.943	16.877	18.268
5000x5000	30.601	33.049	32.668	32.978	46.629	46.845	32.786	33.142	21.579	23.042

Table 24. Processing time for pointer arithmetic and machine code implementations for horizontal and vertical mirror transformations (Intel® Core™ i7-4700 MQ Processor 2.4 GHz).

Image Dimensions (px)	Processing Time (ms)									
	HMIRROR-PTR		HMIRROR-ASM64		HMIRROR-ASM64V2		VMIRROR-PTR		VMIRROR-ASM32	
1000x1000	0.641	0.766	0.643	0.763	2.926	2.976	0.638	0.784	0.415	0.515
1500x1500	1.739	1.983	1.732	1.987	6.602	6.698	1.739	2.049	1.277	1.505
2000x2000	3.162	3.555	3.161	3.667	11.745	11.879	3.101	3.499	2.379	2.709
2500x2500	5.085	6.419	5.051	6.404	18.366	19.321	5.013	6.401	3.906	4.787
3000x3000	7.413	9.617	7.452	9.775	26.465	27.781	7.506	10.031	5.811	7.399
3500x3500	9.347	10.565	9.341	10.295	36.005	36.397	9.462	10.941	7.286	8.169
4000x4000	12.306	13.535	12.233	13.487	47.061	47.342	12.274	13.427	9.402	10.445
4500x4500	15.463	17.044	15.769	17.287	59.592	61.488	15.541	18.419	12.713	13.801
5000x5000	19.115	20.586	19.136	22.236	73.505	74.451	18.999	20.492	13.469	15.324

Table 25. Processing time for pointer arithmetic and machine code implementations for horizontal and vertical mirror transformations (Intel® Core™2 Quad Q9550 CPU @ 2.83GHz).

Image Dimensions (px)	Processing Time (ms)									
	HMIRROR-PTR		HMIRROR-ASM64		HMIRROR-ASM64V2		VMIRROR-PTR		VMIRROR-ASM32	
1000x1000	1.463	2.113	1.553	2.148	1.888	2.653	1.547	2.142	1.529	2.179
1500x1500	4.214	5.773	3.926	6.521	4.554	5.820	3.988	5.715	4.309	5.988
2000x2000	8.059	10.544	7.798	10.112	8.378	12.796	7.614	11.050	7.576	10.100
2500x2500	12.249	18.386	12.210	17.144	15.598	19.888	15.093	19.732	12.169	17.283
3000x3000	18.371	26.424	18.254	25.314	18.089	21.973	18.000	26.796	16.934	24.155
3500x3500	25.399	28.143	25.372	28.172	26.256	28.777	25.586	28.207	25.215	27.929
4000x4000	33.197	35.818	33.543	36.431	34.125	36.308	33.323	35.600	33.212	35.909
4500x4500	41.773	44.704	41.577	44.345	42.924	45.814	41.888	44.578	41.490	44.210
5000x5000	51.507	54.434	51.270	54.754	52.206	55.159	50.988	54.267	51.219	54.305

Table 26. Processing time for pointer arithmetic and machine code implementations for horizontal and vertical mirror transformations (Intel® Core™ i5-3470 CPU @ 3.20GHz).

Image Dimensions (px)	Processing Time (ms)									
	HMIRROR-PTR		HMIRROR-ASM64		HMIRROR-ASM64V2		VMIRROR-PTR		VMIRROR-ASM32	
1000x1000	0.606	0.696	0.604	0.694	1.957	2.086	0.600	0.693	0.366	0.449
1500x1500	1.606	1.858	1.603	1.855	4.409	4.709	1.581	1.836	1.224	1.431
2000x2000	2.931	3.350	2.919	3.345	7.854	8.342	2.886	3.309	2.270	2.624
2500x2500	4.502	5.101	4.481	5.101	12.372	12.995	4.487	5.088	3.654	4.127
3000x3000	6.428	7.274	6.421	7.257	17.832	18.650	6.491	7.365	5.199	5.872
3500x3500	8.791	9.892	8.794	9.889	24.059	25.362	8.835	9.931	7.004	7.930
4000x4000	11.504	12.855	11.523	12.825	32.211	32.974	11.383	12.669	9.116	10.182
4500x4500	14.493	15.930	14.519	15.881	40.634	41.401	14.524	16.026	11.831	13.105
5000x5000	18.106	19.795	18.029	19.754	50.238	51.022	17.989	19.687	14.437	15.586

As with the case with 90-degree rotation, these implementations do not use the mapping offsets. Results show a significant speed up compared with the proposed implementations. Both

pointer arithmetic and machine code implementations give better results than results given in Tables 1-8. It should also be mentioned that, with in Tables 19-26, the HMIRROR-ASM64 machine routine provides significantly better results than machine routine HMIRROR-ASM64V2, which prove the point of the discussion given in section 5. Also, the efficiency of these procedures become increasingly apparent with a growing number of pixels.

The implementation of the new method based on auto-generated machine instructions is the most complex. It exploits the dynamic binding of the auto-generated machine instructions and Units which contain the procedures which are being executed. This mechanism simply connects the static code with the auto-generated code. In fact, this is indeed at the heart of the modification of new method, since the loading of the transformation offsets from memory and their usage, is replaced with automatic generation of the machine instructions and its execution. Since this implementation is somewhat complex and has limitations explained in section 5, processing time results are given only for one image. These results are shown in Table 27.

Table 27. Comparison of the processing time for machine code implementations of the proposed approach and auto-generated machine code implementation (AMD Athlon™ X4 840 Quad Core Processor 3.1 GHz).

Image Dimensions (px)	Processing Time (ms)			
	STANDARD2-ASM32	STANDARDOPT-ASM32	STANDARD-ASM64	AUTOGEN
1756x2273	13.72803	13.11019	13.52518	22.95683

Results show that implementation based on auto-generated machine code is less efficient than standard machine routines despite the fact that it requires one less memory access. The possible explanation for this lies in CPU architecture, since new generation CPUs with large hash systems will give better results with the standard routines.

The new ultra-fast image transformation was developed as part of a real-time OCR system designed for the needs of the “Nikola Tesla Museum” in Belgrade. Therefore, original Nikola Tesla documents was used for demonstration of the performance. Examples of these documents are shown in the Appendix.

7. Conclusions

In this paper a general ultra-fast approach for geometrical image transformations, developed as a part of a real-time OCR system, is presented. Section 2 describes related literature, including work used for evaluation of the proposed method performance. Section 3 gives the detailed mathematical background of the linear transformations including transformation matrices for spatial image transformations. In Section 4 the architecture of the proposed approach is presented in detail, including the optimization of the procedure. The central concept is pre-computed lookup tables to perform the fast image transformations. The crucial optimization step is a linear image representation which ensures a direct memory access. Section 5 gives optimized implementations

of the proposed approach using pointer arithmetic and highly optimized low-level machine code. Beside the various implementations of the new method, optimized procedures for 90-degree rotation, and horizontal and vertical mirroring are also provided. Additionally, a modified variant of the new method based on auto-generated machine code is presented. Section 6 shows an extensive set of numerical results from the perspective of time complexity for several PC machines. Comparative results for image rotation and horizontal and vertical mirror transformations are provided, including analyses of the new method's advantages and drawbacks. The proposed approach provides constant processing time for all transformations regardless of the type of transformation, achieving processing times of below 100 ms, even for very large images. Compared with the classical procedure for image rotation, the new method is 50-60 times faster, and around 10-20 times faster than FPGA implementation. It performs faster than algorithms for forward and inverse rotation, and gives the same results for both types of image rotation algorithms. Taking the various implementations of the new method into consideration, the optimized machine routines give stable results on most PC machines. The specialized machine code implementation for 90-degree rotation performs worse than the new method implementations, however specialized machine routines for horizontal and vertical mirror perform better. The implementation based on the auto-generated machine code proved to be dependent on the CPU architecture type and, in this case, gives worse results than the other implementations of the new method. The method achieves the speed up at the expense of memory usage, since lookup tables have to be computed at run time and stored in memory for the duration. Since the method is part of a novel real-time OCR system, future work will be focused on automation of the OCR system, improving its performance and further optimization of the complete system, including this sub-system and its various implementations.

Acknowledgments

This paper is supported by the Ministry of Education, Science and Technological Development of the Republic of Serbia (Project III44006-10), Mathematical Institute of Serbian Academy of Science and Arts (SANU), and Museum of Nikola Tesla (providing original typewritten documents of Nikola Tesla).

REFERENCES

Ashtari, A.H., Nordin, M.J., Kahaki, S.M.M., 2015. Double Line Image Rotation. IEEE Transactions on Image Processing. 24 (11).

Bensaali, F., Amira, A., Uzun, I. S., Ahmedsaid, A., 2003. An FPGA implementation of 3D affine transformations. Proceedings of the 2003 10th IEEE International Conference on Electronics, Circuits and Systems ICECS 2003.

Berthaud, C., Bourennane, E., Paindavoine, M., Milan, C., 1998. Real time image rotation using B-spline interpolation on FPGA's board. 9th European Signal Processing Conference (EUSIPCO 1998).

Bourennane, E., Milan, C., Paindavoine, M., Bouchoux, S., 2002. Real Time Image Rotation Using Dynamic Reconfiguration. Real-Time Imaging. 8 (4), 277-289.

Cao, Y., Wang, S., Li, H., 2003. Skew detection and correction in document images based on straight-line fitting. Pattern Recognition Letters. 24 (12), 1871-1879.

Chang, H., Fitzpatrick, J.M., 1990. Geometrical image transformation to compensate for MRI distortions. Proc. SPIE 1233. Medical Imaging IV: Image Processing. 116-128.

Chen, C., Ni, J., Shen, Z., 2014. Effective Estimation of Image Rotation Angle Using Spectral Method. IEEE Signal Processing Letters. 21 (7), 890-894.

Cheng, H., Wan, Y., 2015. A new image rotation approach using radial basis functions. 2015 8th International Congress on Image and Signal Processing (CISP).

Devich, R.N., Weinhaus, F.M., 1980. Image Perspective Transformations. Proc. SPIE 0238: Image Processing for Missile Guidance. 322-334.

Eldon, J.A., 1988. Image Transformation And Resampling. Proc. SPIE 0914: Medical Imaging II. 609-613.

Evemy, J. D., Allerton, D. J., Zaluska, E. J., 1989. A stream processing architecture for real-time implementation of perspective spatial transformations. Third International Conference on Image Processing and its Applications.

Fan, P., Zhou, R., Jing, N., Li, H., 2016. Geometric transformations of multidimensional color images based on NASS. Information Sciences. 340-341, 191-208.

Fu, X., Wan, Y., 2015. Accurate image rotation using DCT transformation. 2015 IEEE Advanced Information Technology, Electronic and Automation Control Conference (IAEAC).

Giulieri, A., Nolibe, G., Richon, C., 1988. Image Transformations Using Spline Functions. Proc. SPIE 0860. Real-Time Image Processing: Concepts and Technologies. 82-89.

Hagege, R., Francos, J. M., 2005. Parametric estimation of multi-dimensional affine transformations: an exact linear solution [image recognition applications]. IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '05).

Kapoor, R., Bagai, D., Kamal, T.S., 2004. A new algorithm for skew detection and correction. Pattern Recognition Letters. 25 (11), 1215-1229.

Kovalchuk, A., Peleshko, D., Navytka, M., Sviridova, T., 2011. Using of affine transformations for the encryption and decryption of two images. 2011 11th International Conference The Experience of Designing and Application of CAD Systems in Microelectronics (CADSM).

Li, K., Dan, T., 2013. Fast rotation and correction of image algorithm based on local feature. 2013 International Workshop on Microwave and Millimeter Wave Circuits and System Technology (MMWCST).

Liu, D., Yin, S., Liu, L., Wei, S., 2013. Affine transformations for communication and reconfiguration optimization of loops on CGRAs. 2013 IEEE International Symposium on Circuits and Systems (ISCAS).

Lopes, F., Ghanbari, M., 2002. Three-dimensional overlapped spatial transformations for motion compensation. 2002 International Conference on Image Processing.

Lucchese, L., 2001. Estimating affine transformations in the frequency domain. 2001 International Conference on Image Processing.

Mahata, K., Ramakrishnan, A.G., 2000. A novel scheme for image rotation for document processing. International Conference on Image Processing. 2, 594-596.

Mondal, P., Biswal, P.K., Banerjee, S., 2016. FPGA based accelerated 3D affine transform for real-time image processing applications. Computers & Electrical Engineering. 49, 69-83.

Nakaya, Y., Harashima, H., 1994. Motion compensation based on spatial transformations. IEEE Transactions on Circuits and Systems for Video Technology. 4 (3), 339-356.

Nuno-Maganda, M.A., Arias-Estrada, M.O., 2005. Real-time FPGA-based architecture for bicubic interpolation: an application for digital image scaling. 2005 International Conference on Reconfigurable Computing and FPGAs (ReConFig'05).

Pei, S., Hsiao, Y., 2015. Spatial Affine transformations of images by using fractional shift fourier transform. 2015 IEEE International Symposium on Circuits and Systems (ISCAS).

Pham, Q., Nakamura, Y., 2015. A New Trajectory Deformation Algorithm Based on Affine Transformations. IEEE Transactions on Robotics. 31 (4), 1054-1063.

Piperakis, E., Kumazawa, I., 2001. Affine transformations of 3D objects represented with neural networks. Third International Conference on 3-D Digital Imaging and Modeling.

Qian, R., Li, W., Yu, N., 1993. High precision rotation angle estimation for rotated images. 2013 IEEE International Conference on Multimedia and Expo Workshops (ICMEW).

Raittinen, H., Kaski, K., 1993. Fractal based image compression with affine transformations. 1993. DCC '93. Data Compression Conference.

Rodrigues, N. M. M., da Silva, V. M. M., de Faria, S. M. M., 2001. Hierarchical motion compensation with spatial and luminance transformations. 2001 International Conference on Image Processing.

Ryu, S., Lee, H., 2014. Estimation of linear transformation by analyzing the periodicity of interpolation. Pattern Recognition Letters. 36, 89-99.

Singh, C., Bhatia, N., Kaur, A., 2008. Hough transform based fast skew detection and accurate skew correction methods. Pattern Recognition. 41 (12), 3528-3546.

Schmalz, M.S., 1993. Spatial transformation architectures with applications: an introduction. Proc. SPIE 1961: Visual Information Processing II. 55-67.

Tan, Z., Zhang, Y., Song, J., 2012. A regional rotation algorithm of video images. 2012 5th International Congress on Image and Signal Processing (CISP).

Unser, M., Thévenaz, P., Yaroslavsky, L., 1995. Convolution-based interpolation for fast, high-quality rotation of images. IEEE Transactions on Image Processing. 4 (10), 1371-1381.

Weeks, A.R., Myler, H.R., Emery, J.D., 1994. Nonlinear image transformations implemented with spatial light modulators. Opt. Eng. 33 (3), 850-855.

Yamashita, Y., Wakahara, T., 2016. Affine-transformation and 2D-projection invariant k-NN classification of handwritten characters via a new matching measure. Pattern Recognition. 52, 459-470.

Yi, K., Joo, J., Kim, K., 2015. Compressed Image Quality Distortion Problem due to Repeated Rotation in Spatial Domain and Its Solution by Changing Image Information in Frequency Domain. 2015 8th International Conference on Disaster Recovery and Business Continuity (DRBC).

Younes, L., 2006. Combining geodesic interpolating splines and affine transformations. IEEE Transactions on Image Processing. 15 (5), 1111-1119.

Yu, B., Jain, A.K., 1996. A robust and fast skew detection algorithm for generic documents. Pattern Recognition. 29 (10), 1599-1629.

Yu, Z., Dong, J., Wei, Z., Shen, J., 2006. A Fast Image Rotation Algorithm for Optical Character Recognition of Chinese Documents. 2006 International Conference on Communications, Circuits and Systems Proceedings. 1.

Zhu, N., Deng, C., Gao, X., 2016. A learning-to-rank approach for image scaling factor estimation. Neurocomputing. 204, 33-40.

Appendix. Example of processed Nikola Tesla's document using the provided implementations.

SIGMA PI FRATERNITY
PURDUE UNIVERSITY
WEST LAFAYETTE, IND.

May 23, 1916.

Mr. Nikola Tesla,
8 West 40th St.,
New York, N.Y.

My dear Mr. Tesla:

I have discontinued my work at the University and am now in search of employment.

I believe you know something of my qualifications. If you also know of any employer who may require such services as I am able to render you will favor me greatly by putting me in touch with him. I wish to thank you in advance for this favor.

Very truly yours,

Horizontal mirror (HMIRROR-ASM64)

MNT. CXXVI.
K 152

A 03

SIGMA PI FRATERNITY
PURDUE UNIVERSITY
WEST LAFAYETTE, IND.

May 23, 1916.

Mr. Nikola Tesla,
8 West 40th St.,
New York, N.Y.

My dear Mr. Tesla:

I have discontinued my work at the University and am now in search of employment.

I believe you know something of my qualifications. It you also know of any employer who may require such services as I am able to render you will treat me greatly by putting me in touch with him. I wish to thank you in advance for this favor.

Very truly yours,

D. F. Harris

90-degree counter-clockwise rotation (90DEG-ASM64)

[1]**Vladan VUČKOVIĆ** was born in Niš, Serbia, in 1970. He received the B.E. degree in electrical engineering from the University of Niš, Faculty of Electronic Engineering, Niš, Serbia, in 1994, and the M.Tech. and Ph.D. degrees in electrical engineering and computer science from the, University of Niš, Faculty of Electronic Engineering, Niš, Serbia, in 1997 and 2006, respectively. In 1995, he joined the Computer Department of Faculty of Electronic Engineering, University of Niš, as a Researcher, and in 2003 became an Assistant, in 2007 an Assistant Professor, and became an Associate Professor in 2012. His current research interests include theory of games, artificial intelligence, machine programming and optimization, information security, and 3D modeling, simulation and virtual reality. Dr. Vučković is a member of the International Computer Games Association (ICGA) since 2008. He is a leader of the national project "Advanced methods in 3D modeling and computer simulation of the original patents of Nikola Tesla" since 2009. He is Pupin prize (1995) and Tesla prize (2012) winner.

[2]**Boban ARIZANOVIĆ** was born in Surdulica, Serbia, in 1991. He received the B.E. degree in electrical engineering from the University of Niš, Faculty of Electronic Engineering, Niš, Serbia, in 2015. Since 2016, for the purpose of academic research and faculty projects working on development of the real-time OCR system. Since 2016, actively researching the predictive modelling approaches for real-world examples. His current research interests include image processing, pattern recognition, artificial intelligence, machine learning, optimization problems, digital signal processing, and information security.

[3]**Simon LE BLOND** was born in Bath, UK. Simon obtained his PhD in 2011 in power systems from the University of Bath and since 2012, has been an academic Bath's department of Electronic and Electrical Engineering. Simon's primary research area is using AI for Electrical Power System protection and optimization, but he collaborates with academics across the world in a number of other areas.